Australian Government
Department of Defence
Defence Science and
Technology Organisation

Debugging and Logging Services for Defence Service Oriented Architectures

Michael Pilling

Command, Control, Communications and Intelligence Division

Defence Science and Technology Organisation

DSTO–TR–2664

ABSTRACT

While often thought of as a "Dark Art", debugging is nevertheless a necessary part of fielding quality computing systems which can and should be done systematically. Service Oriented Architectures (SOAs) show great promise but also represent one of the most challenging environments in which to debug system services. In addition to all the issues of distributed and paralled debugging, SOAs introduce the complexity of significant parts of one's programs being provided by others. This paper examines the features of SOAs that complicate debugging and shows how integrated logging is an essential part of finding the cause of service failures. We draw on Agan's work in developing systematic strategies for debugging to generate system features that are necessary or helpful for debugging in an SOA environment. These are in turn used to specify requirements for a debugging system integrated into the fabric of the SOA. We argue that deep integration is necessary to produce significant debugging efficiency improvement in an SOA environment and provide some recommendations for Defence in this area.

APPROVED FOR PUBLIC RELEASE

Published by

DSTO Defence Science and Technology Organisation
PO Box 1500
Edinburgh, South Australia 5111, Australia

Telephone: (08) 7389 5555
Facsimile: (08) 7389 6567

© *Commonwealth of Australia 2012*
AR No. AR-015-224
February 2012

APPROVED FOR PUBLIC RELEASE

Debugging and Logging Services for Defence Service Oriented Architectures

Executive Summary

This document proposes an integrated distributed debugging and logging service for Defence's Service Oriented Architecture. It argues for them on the basis that:

- Debugging SOA specific issues requires integrated debugging hooks within SOA components.
- Message and event logging with vector timestamps will allow programmers fixing bugs to avoid many spurious symptoms and get a clearer picture of the problem at hand.
- Logging of application and component success or failure is essential to build up a statistical picture on which to quantify the reliability of SOA-based subsystems as a basis for fielding such subsystems for deployment.

The report also provides a software specification for such an SOA debugger. The requirements for this debugger and associated logging system are derived specifically from debugging strategies that are known to work and the information requirements that support them.

It makes several recommendations chief among them being that whatever SOA infrastructure Defence chooses, it must be flexible enough to allow Defence to modify key components in order to add functions supporting debugging and adequate logging.

THIS PAGE IS INTENTIONALLY BLANK

Author

Michael Pilling
C3ID

Michael Pilling completed a Bachelor of Science degree with honours in 1987 and a Ph.D. in Computer Science in 1996 at the University of Queensland, Australia. Michael's specialities are distributed and real-time systems, job scheduling, formal specification and program correctness, criticality management, and the calculus of time. His current interests include Software Reliability Engineering, Failure as a fundamental construct in usable and effective systems, Virtual Synchrony and its application to synchronous group communication, performance engineering of computer systems, and graceful degradation of systems in the face of failure and overload.

THIS PAGE IS INTENTIONALLY BLANK

Contents

THIS PAGE IS INTENTIONALLY BLANK

Glossary

Business Process An activity which provides some business value that may be embodied in **services** and **jobs**. While services and jobs are programs and are executable by a machine, a business process also encompasses any human activity involved in achieving the business result.

Causal Order A partial order of events in which $a \Rightarrow b$, read event a is causally before event b, means that the result at b *may* be a consequence of the results or behaviours at event a. $a \Rightarrow b$ if a occurs before b in the same process; or if a occurs in a process that sends a message to a second process, and b occurs in the second process after it has received the message from the first; or transitively i.e. if $a \Rightarrow b$ and $b \Rightarrow c$ then $a \Rightarrow c$ also. If there is no causal relationship between a and b in either order, then they are said to have executed in parallel. While they may or may not have been executed simultaneously, their execution order is definitely unimportant to the results of the overall calculation and so could safely be executed simultaneously on different processors.

Choreography The composition of services into a **job** in a dynamic and distributed manner where no party has the script but each may have some rules about how to interact. In many ways choreography is data-driven rather than control-driven. It can be argued that each execution of choreography generates its own script as it runs and so each *run* effectively results in a script that can be derived from a subset of its **trace**. The logging system must distinguish between the different actual event orderings evident in each run of a Choreography. By using a causal order analysis of each such trace, the system can distinguish between different event orderings that occur due to chance timing and those that occur due to fundamental causal orderings dictating the order of execution. This allows the traces to be grouped together by their fundamental differences, not by their apparent (chance) ones. Contrast with **orchestration**.

Enterprise Service Bus (ESB) A distributed component of most SOA systems through which all service calls are routed. The ESB may, depending on the system, provide services such as service name resolution, routing, monitoring, external logging, timestamping, maintenance of correlation IDs, mapping and translating data, security, reliability enhancement (e.g. failover, voting), etc. As a focal point on each node, it is a natural place to include extra functionality that enhances debugging and/or implements logging capability.

Job A complete unit of work that starts, runs and terminates. It may consist of a single call to a service, or the execution of a single program; however, in an SOA, it will most often consist of a script that composes the behaviour of multiple services to produce an overall operation. This composition is referred to as **orchestration** or **choreography**. Scripts may be hand-written or automatically generated, however in our definition we insist that each textually different[1] script constitutes a different **job** even when two different scripts achieve the same outcome. This is essential for

[1] apart from comments

characterising the behaviour and the reliability of particular scripts which may differ although they do the same thing.

Lamport Clock A sequence vector whose local element is incremented whenever an "operation" is performed locally. It is attached to messages as a timestamp when sent. On receiving such a message, a process or node sets its own Lamport Clock to the component-wise maximum of its local Lamport Clock and the timestamp received in the message. Lamport Clocks are used to provide vector timestamps which in turn allow a **causal order** to be established over events that are tagged with the local timestamp when they occur. In order to ensure that the same vector timestamps always correspond to the same execution order, either the message transport mechanism including the *ESB* must prevent message overtaking even in the transitive sense, or the system must augment vector time stamps with local clock times to produce different **traces** for each overtaking case[Fid98]. Note that an SOA is an ideal environment in which to use vector timestamps because the service nature of data interactions minimises the opportunities for causal dependencies to go unnoticed, for example via untracked back channels such as updates to shared disks. In SOA, each service "guards" its own data so causal relationships are made explicit.

Logging The facility to record information about the execution of a process, or the process of recording same. In the armed forces, this includes the records made by humans to record the changes in their environment and their decisions made. In the case of logging of computer processes, *self* logging refers to records submitted by the program itself (at the programmer's discretion) to record particular events, changes in state or variable attributes that indicate the progress or failure of the program. It is often used to externalise and record unusual or important internal program states and the program's reaction to them. *Sentinel* logging refers to records made about a process's behaviour by an external observer, often the system. Sentinel logging is used to record information about the entire process or its behaviour, such as whether it completed normally or failed, its overall resource usage, run-time etc and is often performed after the process itself terminates. Both types of logging usually includes a severity level to allow the system or human readers to filter the level of detail they wish to observe or record.

Orchestration The composition of services using a controller and a script that is determined before execution. Many controllers can orchestrate different jobs simultaneously. The orchestrated composition itself can form a **service** or a **job**. Contrast with **choreography**.

Quality of Service (QoS) "In the field of computer networking and other packet-switched telecommunication networks, the traffic engineering term quality of service (QoS) refers to resource reservation control mechanisms rather than the achieved service quality. Quality of service is the ability to provide different priority to different applications, users, or data flows, or to guarantee a certain level of performance to a data flow. For example, a required bit rate, delay, jitter, packet dropping probability and/or bit error rate may be guaranteed. Quality of service guarantees are important if the network capacity is insufficient, especially for real-time streaming multimedia applications such as voice over IP, online games and IP-TV, since these often require

fixed bit rate and are delay sensitive, and in networks where the capacity is a limited resource, for example in cellular data communication." [Qos]

Implementation of quality of service requires all components in the "value chain" to respect QoS and may require resources given to some activities to be throttled (limited) so that other activities can meet their performance requirements.

Run A single execution of a particular **job**.

Service A software component and callable end point that provides a logically related set of operations, each of which perform a logical step in a business process. A service normally stays alive waiting to be called and performs some work on behalf of a job or higher level service when called to do so.

Service Level Agreement (SLA) A "contract" between a service provider and its user guaranteeing certain levels of performance, including measures such as response time, availability and reliability. An SLA may apply at the level of a particular service, or a composition of services, or indeed the entire system itself.

Trace A trace of a **run** is the complete sequence of log records pertaining to that particular execution sorted into causal order. It may be filtered to a particular logging level.

Vector Timestamp see **Lamport Clock**.

THIS PAGE IS INTENTIONALLY BLANK

1 Introduction

CIOG group have indicated their intention to move to a Service Oriented Architecture (SOA) based whole of Defence platform[CIO10] referred to as the Single Information Architecture. The concept envisages multiple Defence applications being built from common components across divergent networks integrating new and legacy systems.

This document looks at what can be done to facilitate the creation and maintenance of the most bug-free, reliable and stable information technology environment possible based on SOA using a broadened view of traditional debugging and logging. This requires that our view of correctness is correspondingly wider and includes concepts of performance, security and utility beyond pure algorithmic correctness. While this paper will talk about uses of logging and debugging in quantifying the reliability of a set of SOA components or applications, detailed discussion of such reliability engineering is left to a following paper.

2 What's different about an SOA?

There are many definitions of what constitutes a Service Oriented Architecture (SOA) and the one chosen by CIOG is:

> Service Oriented Architecture (SOA) represents an architectural style that aims to enhance the agility and cost-effectiveness of delivering IT capability within an enterprise while simultaneously reducing the overall risk and maximising the organisational investment in its IT capability. It accomplishes this by encapsulating technical capability as one or more business services that are used and re-used throughout the enterprise. SOA supports service-orientation through the realisation of the strategic goals represented by service-oriented computing. For example, some key SOA goals include risk reduction, agility, and leveraging existing technology investments.[CoA11]

However the following one:

> SOA is an architectural paradigm for dealing with business processes distributed over a large landscape of existing and new heterogeneous systems that are under the control of different owners.[Jos07],

is also highly relevant to the Australian Defence context because of Defence's need to integrate many legacy systems, to interoperate with other forces and increasingly to interoperate with other Australian Government Departments, both State and Federal, as well as with businesses and NGOs for counter-terrorism operations and civil emergencies.

Other relevant definitions emphasise platform independence and loosely coupled interoperability[SOA06]; visibility, interactions and their effects[Oas06] and implementation diversity and whole of life-cycle management[NL05].

Web Services, in and of themselves, are neither necessary nor sufficient to create an SOA. Other distributed technologies such as message queues can be used to build an SOA

and it is the organisation of interfaces around self-contained business process steps that most strongly distinguish SOAs from other distributed architectures which are usually implementation centric. It is these relatively course-grained independent business process steps that lend themselves to composition into many different business applications. SOAs are focused on creating end user business value.

Nor do distributed objects such as those commonly provided by CORBA constitute an SOA[Jos07]. Experience shows that such remotely accessed objects are code-centric rather than business process-centric and generally result in complicated, highly coupled applications with lots of dependencies that will not scale.

The focus on business processes and value affects the applications and structure of SOAs at every level. In particular, SOAs that have any long term existence almost inherently end up being heterogeneous implementations as more of the organisations existing infrastructure is linked into the SOA. While the organisation may seek to constrain this heterogeneity, and this is a useful goal, eventually a merger or partnership interaction will force the SOA to accommodate foreign technologies. Likewise any SOA that is deployed in real world operation will eventually end up with multiple versions of the same service co-existing as upgrades occur.

To meet the challenge of providing a usable and reliable yet diverse operating platform for Defence, system specialists will have to integrate components, including legacy components, into working applications that can provide semantic, operational and timing guarantees to their users or other higher order components.

This paper explores the use of logging and advanced debugging which we believe are essential technologies if the goal of achieving a working Defence platform is to be achieved.

3 Purposes of integrated debugging and logging support

All of the Australian Armed Services have a requirement that their officers record logs of their activities, orders and of events for the purposes of process improvement, historic and judicial reasons[Arm07]. The needs of this type of logging diverge from the needs of logging for SOA system maintenance. However, in the spirit of SOA some components of the two systems could be shared, most likely quick paths to persistent storage, replication, non-repudiation, archiving/expiry and remote access. That noted, we shall concentrate on the latter purpose and type of logging in this paper.

Debugging is usually considered purely as a programming issue or as something that is used only during development, program maintenance and upgrade. Similarly, a logging system is often considered in isolation. However, a fully developed and integrated debug, test and logging environment can act synergistically to provide useful services throughout the life of the system, including the full life cycle of individual applications. These services include:

Detection Built in code and sanity checks can provide early detection of errors in the code that produce incorrect results. This contributes to data hygiene preventing the

propagation of errors that would pollute data sets and lead to downstream errors. Such early detection can also simplify the debugging task of localising the error as the chain of problems is shorter when errors are caught quickly.

Debugging By this we mean the localisation, identification, characterisation, management and preferably correction of faults or defects in the program. In practice, this can mean manipulating the program through a breakpoint style debugger and stepping through the program in slow-time. It's important to note that in some cases when the fault is identified to lie in uneditable code such as program libraries, or outsourced software services debugging is limited to characterisation of the fault, reporting it to the software or service provider and development of work-arounds and management strategies. This will often be the case for Service Oriented Architectures, where even if the service is provided by the same organisation it may be another section that maintains the code. Even in such instances, localising, diagnosing and documenting the problem as occurring in code supplied by others is essential for being able to strongly demand correction of that code and establishing appropriate work arounds.

Documentation Given the demarcation disputes that can arise with externally sourced code and services, an increasingly important function of an integrated debugging and logging system is to document faults in order to provide proof to external parties that rectification is needed. Well built debuggers can provide a significant amount of this documentation.

Test and Qualification Well organised debugging systems supporting software engineering methods such as Cleanroom Software Engineering[MDL87] and Extreme Programming[Bec99] create test suites that are used for preemptive debugging. These test suites can be migrated into production systems to provide ongoing health checks and early fault detection[DH04][1]. These will usually be hardware faults, but also software faults exposed by new uses embodied in new applications composed from existing software components and by component or client software upgrades[DH04]. Results of automatic testing can be used to certify software component reliability levels and thus qualify them for particular deployments and uses[PMM93]. With proper test and development process management, reliability levels can be expected to increase exponentially as a function of the number of tests run[PMM93] shortening the time to qualification and hence deployment. Another form of Qualification is to test the results of new software against the results of existing software to prove that the new work is substitutable for the old. The most familiar instance of this is regression testing to prove that upgraded or debugged software does not reintroduce old bugs that have previously been removed. In an SOA, it may be possible to run the new software in a parallel shadow configuration and compare each of its outputs with the existing software to demonstrate that its responses constitute a proper superset of the existing software's responses. This can be applied to software that comes from an entirely new source, as well as upgraded software.

Sequencing, Trace and Audit One of the major complicating factors of distributed systems is the potential for inter-process communication messages to be reordered

[1] As such these test suites need to be part of the deliverables provided by contractors.

as they pass through the network. Left unchecked, this combined with the potential parallel execution of asynchronous calls, leads to a combinatorial explosion of potential computation states. There are many ways this state space can be collapsed into a far more tractable one. However, since not all applications or systems go to any effort to constrain this state, it is essential that an integrated debugging environment record the sequence and to a lesser extent the timing of message departures and arrivals.[2] This allows the true sequence of events that led to the manifestation of the bug to be seen.

Once SOAs reach the point of using composite applications, which is a major goal of CIOG[CIO10], it becomes important to be able to trace the dynamic calling tree of an invocation across the many services, machines, and governance and security domains that constitute the business process for reasons such as cost accounting, resource provisioning, security auditing and fault characterisation. In addition, it allows the human debugger to see the full call and data trace to be able to isolate the bug. This involves recording the particular instance of each service that is invoked because the fault that arises in that service may be an artefact of the entire history of a particular instance. Thus it is important to correlate one service's call to another service's invocation.

Replay and Simulation Being able to replay all or part of a high level system invocation assists the programmer in comprehending and isolating a fault. Equally this type of facility can assist people in distributed algorithm development where runtime behaviour may differ from that expected, even when it is not incorrect. For instance, such a facility also allows people in various roles to be trained in aspects of the system. Another important use of replay is to allow automatic testing of debugged modules by effectively providing test harnesses and test data for them.

Monitoring Given the high level of diversity and extent of SOAs, a significant part of managing an SOA is to constantly monitor its health, both for new interaction bugs arising through activation of a previously untraversed execution sequence, and to detect transient overloads, service failures, bottlenecks and long term capacity constraint issues. Where SLAs are entered into, data must be collected to prove compliance and to feed into adaptive management systems that control the overall system to ensure compliance. Notably, it is statistically possible for an SLA to be met on average, but not for particular users or service instances so it is important to ensure SLA promises are delivered to all clients. Thus the debugging and monitoring infrastructure can also be important in managing failover and recovery at a macro level. A primary function of monitoring is to visualise the dynamic behaviour of the SOA in a timely and exception/fault focused manner. Another useful output of general monitoring of the SOA is to calculate $call^{-1}$, that is the "who calls me" function: which can be important given that service providers are otherwise generally unaware of who calls them. SOAs are intended to invite unexpected uses, and service upgrades are improved if users are asked what they might want.

System, subsystem and component reliability can also be monitored to provide measures rather than estimates of software properties for service qualification.

[2]Sequence can be calculated accurately, but timing may be flawed due to clock skew, etc.

Providing a realistic sandbox A good debugging and bug prevention system will provide a realistic, effectively isolated environment as a "sandbox" to allow the development and thorough testing of new code and services without risking the stability of the business environment. This can be provided by separate hardware and software or by strictly partitioning[3] existing infrastructure e.g. by allocating 10% of disk space to test and development in a separate disk partition, or by allocating 20% of CPU time or communication channel bandwidth to the sandbox activities. Strict partitioning is a more difficult but more general and representative challenge. In either case the governance[4] system needs to be able to provide a gradual path out of the sandbox into production.

3.1 Debugging Strategies

Debugging is usually regarded as a somewhat dark art, but experienced software and hardware engineers such as David Agans insist that quick and thorough debugging is a matter of systematically applying specific strategies. Agans[Aga02] lists many strategies under nine overarching idioms to apply to isolate, characterise and ultimately fix any fixable bug.

Here we list several of the detailed strategies and indicate how an integrated SOA debugger and quality assistant might aid in each strategy:

Know the fundamentals By recording and providing a history, normal system and component behaviour is available as a baseline to compare to a specific failed instance. Divergences can therefore be noticed early and the likely area of the fault localised before the programmer is dealing with secondary (consequential) symptoms.

Look up the details Original documentation should be available to guide programmers and debuggers as to the intended use of the software, its components and their operational characteristics. By providing this online, it will be available to programmers no matter where they happen to be.

Do it again Allow scripts to be written for the repeatable running of the same code in the same environment. This allows the cause to be narrowed down and to eventually tell if the bug has been eliminated.

Start at the beginning A full history of the running instance and its interactions will make many unexpected or otherwise unconsidered program inputs visible.

As far as possible, the system should provide an isolated testing environment starting the application instance off with a purely clean slate. Some iterators may be able to be "parallelised" to start each iteration off in a clean state, or services being called can be booted from scratch before the test run. Once any bugs arising in this clean environment are fixed, gradually more and more of the real-world (production)

[3] Preferably with hardware support.

[4] Governance is the set of operational procedures implemented by the combination of humans and systems which controls the changes made to the SOA system to ensure its stability and fitness for purpose.

environment should then be added to the test runs -e.g. interacting with long running services to mimic production runs.

Record everything and find the signature of intermittent bugs Having records of correct running instances, and also of failed ones allows the programmer to discover what is always correlated with the bug's manifestation and what isn't.

Never throw away a debugging tool Storing related debugging tools, such as programmer written test scripts, with the software gives debuggers the same tools as the original programmers.

See the failure Being able to visualise the operation of the [sub-]system, its components and messages can assist the programmer in finding the bug and thus linking it to a real cause.

See the details Being able to see the actual interaction details of the many components in each instance can help localise the problem.

Build instrumentation in Having a multi-level, run-time filtered logging facility can allow the right amount of detail to be logged and later examined.

Watch out for Heisenberg Heinsenberg showed that any instrumentation will distort the behaviour of a system it measures.[5] The weight of the debugging and logging infrastructure should not overwhelm the system nor distort its behaviour to the point where results are misleading or substantially different from what would have occurred without the monitoring. This means the infrastructure should be as light weight as possible and turning the debugging or logging system on should not change the order in which messages are delivered compared to its off state.

Use easy-to-spot test patterns The debugger should generate, or provide from a library, test patterns that produce easily visible behaviours. Many of these test patterns may be provided by the original developers.

Fix the bugs you know about The system should allow the programmer and others to maintain a prioritised list of bugs filterable by sub-system. The bug reporting and ticketing system should be integrated with the debugging aid. Previous tests and their expected results should be recorded so that automatic regression testing can be performed once a bug fix is installed.

Change one thing at a time The debugger should be integrated with a change control system.

Determine what has changed since the last time it worked Timestamps on traces and successful and unsuccessful completions should be cross referencable with code, thus integrated version control would be helpful.

Start with the bad The system should implement multiple validation checkpoints throughout the operation of a run-time instance. The system should encourage run time testing of assertions and invariants and facilitate these being evaluated against the

[5] Increasingly people are questioning whether Heisenberg did, in fact, claim this but we are following Agans' nomenclenture, and the weight of instrumentation argument is still valid.

SOA state, e.g. message ordering properties, system load etc. The system should draw the programmer's attention to the earliest divergent part of the run-time trace.

Compare it with a good one The system should assist the user in categorising trace instances as good or bad and finding correlations between the categories and data inputs and input orderings. It should also suggest further subcategories based on trace correlations.

Correlate events The system should record as far as possible the sequence of events that did occur, and assist in determining which events did occur by chance ordering and which are ordered in a particular way by inherent properties of the algorithms.

Understand that audit trails are also good for testing Use configuration control tools and failure logs to detect which revision introduced the bug.

Write it down! When a program fails, the system should get the end user or tester to record the circumstances in which the failure occurred and any salient comments.

Check if it is really fixed The system should provide extensive monitoring of recently fixed (or developed) code, and run regression tests intensively. Stable components should still have the occasional regression test run and be monitored in general.

Fix the cause The system should facilitate the retirement of old code, both for economic efficiency, and to prevent deprecated code causing problems for components which call it.

4 A specification for an integrated SOA debugging and health system

For clarity, we first introduce a concept for sorting trace evidence that is used in the wider specification.

4.1 Category Partitioning Specification

A category partition is a partitioning of traces into several distinct sets so that each trace is a member of exactly one set of that category partition. For example you could partition over the sequence of inputs (their order and value). If there was only one input, a natural number, the sets of traces could be labelled by that number. Equally a different category partition of the same traces might distinguish on the basis of giving a correct or incorrect output, and another on the basis of clean program termination or abnormal termination (e.g. array out of bounds) or hanging.

An important category partition in distributed debugging will be the order of events based on the causal order generated by the Lamport Clock timestamps. In other words the partitions will be based on the order of events except that the order of mutually parallel events are viewed as irrelevant; that is any order among them is seen as identical. In this way, only substantially different category partitions for traces are produced.

The system should automatically generate several category partitions such as input values, output values and event (causal) order and give them and their sets default names. The system must allow programmers to create and name their own category partition, as well as rename system produced ones.

Once partitions are established, the intersection of sets from distinct category partitions can be used to create sets of traces corresponding to particular run time properties, most obviously "working" and "buggy" but also more specific subcategories of these.

4.2 Overall Specification

A system to minimise the work required to debug SOA services, be they self-contained or composite, and which assists in quantifying and raising the reliability of these services[6] would have as many of the following integrated properties as possible. It is through the combination of these well integrated properties that the maximum benefit is obtained.

Bug reporting and tracking system This is important, not only to record the symptoms and circumstances of all bugs, but also to eliminate as many bugs as possible, leaving only the symptoms of the hardest uncorrected bugs. This reduces noise allowing the true error signals to become visible leading to clearer characterisation of the error. End users should be queried when any run fails so they can describe what happened and also asked to point out anything novel about the run or data. The bug tracking system must link bugs to particular versions of each service and orchestration or choreography scripts if applicable, moreover it should link bug reports to particular runs and the logs from those runs if they exist.

Distributed Debugger An SOA specific distributed debugger should not only allow remote invocation and control of software artefacts as is common for such software but also:

- be able to see the specific details of each message/component interaction by accessing either log files or interacting with the ESB(s), including which of perhaps many suitable services was actually invoked.
- insert or alter messages going through ESB(s).
- specify messages, including by using wildcards, as breakpoints when they reach an ESB.
- provide a debugging flag in the message format and be debugging flag aware so that when a message arrives from a *run* being debugged, it causes the receiving service to invoke its debugger interface and set the debugging flag for all messages that reply to that message/run.
- allow the programmer to work with category partitions to isolate traces corresponding to particular symptoms and compare them to the traces that are correct.

[6] We leave the detail of how to improve SOA reliablilty to later paper(s), however where it is obvious that some particular data is necessary to do that we include it in this specification.

- insertion of automatically generated code to log variable values and programmer messages so that these will appear in future traces[7].
- As a second order requirement, to be able to visualise the infrastructure that the job/task/etc is using, and the interactions of the various components during the run.
- Allow the programmer to get a whole of system view of the runtime environment and the software components used (even if they can only appear as black boxes).

ESB that integrates vector timestamping, correlation IDs, and QoS In an SOA the ESB or other middleware is the logical place to intercept interprocess communications and to augment them. This permits vector timestamps to be used even with "black box" software that is vector timestamp unaware, although it means treating the "black box" processing in between message interactions with such processes as a single event.

The ESB and ESB interface code in services must also maintain correlation IDs so that all the threads of a particular run of a job can be correlated together. Without such correlation, it is very difficult to associate failing service executions with their callers and the data that was passed to them.

The ESB should have a debugging interface so that authorised programmers can get the ESB to hold a received message so that they can use the debugger to examine and optionally modify it before it is forwarded to its destination(s).

The ESB, like other components, must honour QoS if the SOA is to deliver performance guarantees in its SLAs. A well implemented QoS system will allow the system to quarantine testing and development and so run it on production hardware without risking the reliability or performance guarantees of production jobs.

Job Scheduler At the instantiation of a new job run, the job scheduler needs to assign the run a system-wide unique ID that can be passed through all calls to other services to link them together as a single logical entity. It should create a sentinel log record of the run's commencement time and operating environment, as well as security and accounting information as it invokes the new run. Likewise performance attributes and resource usages of the run on job termination, and at appropriate intermediate waypoints, should be logged by some sentinel process. It should also be able to flag a run as a debugging run to ensure that the local part of the debugger is remotely invoked on a service whenever any code being debugged that calls that service is executed.

Logging Service The logging service must be capable of:

- logging the full history of runs, including those of choreographies (where possible — see next point) and orchestrations, in a distributed manner and then coalesce these fragments into a single log.
- logging the version of a service when invoking a service across a domain boundary. That is, even though the other domain may not provide shared logging in a choreographed interaction, it must at least maintain correlation IDs or behave

[7]Note that we see tracing to be a function of logging more than the debugger in this type of system.

so as to allow their deduction and also identify which version of its software has responded in the interaction. This is essential to allow category partitioning and to allow a service user to identify whether a newly apparent bug is likely to have arisen from changes in their own code or data, or is most likely caused by changes in invoked foreign services.

- tagging each entry with a number of tags to allow later filtering, such as severity, production or test, etc., and the extant software and hardware configuration, etc. Severity of the message should not be the sole criteria on which the level of logging and filtering is determined.
- logging the termination status of each run (normal or abnormal) and also tagging runs that receive user-generated behaviour deviation reports.
- logging both the system and software configuration at the time of the run and the actual hardware and software that the run was executed on, which may only be a subset and can change as the job progresses.

Correlation Engine The correlation engine is used to not only link log entries with correlation IDs into complete traces, but also to automatically sort traces into various category partitions. Each new failure should reference the most recent available successful run if one exists, and list the changes both in terms of code and configuration that have since occurred.

Source Code / Version Control System Including a documentation repository and references to associated tests. Automatically generated orchestration and choreography scripts must also be recorded here so they are reused. Reuse is not only for the efficiency of avoiding regeneration, but also to build up sufficient experience with a script to be able to assess its reliability. The compiler should provide source code alteration so that test code, such as assertions, can be included or omitted (with compiler analysis to ensure there is no change to the substantive code because the test code is side effect free.). Old code that has been superseded should be encouraged to be fully retired by raising alerts over too many active versions. The source code control should monitor code overlap and unintended code and service redundancy.

Unit Test and Regression Test repository While unit and regression tests are now standard software engineering practice some SOA specific issues are worth noting. Unit tests should encourage assertions in production code so that choreographies are halted as soon as they become incorrect: There are two purposes for this:

1. Isolating the bug, drawing the programmer's attention to the problem as close as possible to its cause; and
2. Ensuring other services that would otherwise be called later are neither contaminated with unnoticed bad data, nor caused to fail thereby reducing their own reliability ratings.

The repository should automatically rerun regression tests on code that is in use in an exponential back off manner[8] so that production code is always retested but at reducing intervals. This is to detect unforeseen incompatibilities with other code it

[8] I.e. newer code is tested exponentially more frequently than established code.

does not formally interact with (e.g. code may slow down due to other loads on the system).

Configuration Management System A strong configuration control system and associated governance is needed to control the set of hardware and software allocated for tasks. This not only helps in providing a reliable environment but simplifies fault attribution. An important aspect of a configuration system for SOA would be to provide representative but isolated test areas in which to run code that has not yet reached production, or is being regression tested, i.e. the ability to either duplicate environments or provide strong execution isolation within a production environment is required.

Log Database Not only does this database need to be able to record the log and trace data for each individual run, indexing it by versions and category partitions, etc., it must also be hierarchical in the sense that many SOA runs will consist of the composed runs of subcomponents. As such a single trace of a service run must be accessible as the run of itself in isolation, as part of the run of the service that called it, and as part of the run of the service that called that service and so on.

Replay system The replay system must be able to utilise the *configuration system* to reconstruct an equivalent configuration to that of a specified run. The job can then be rerun in debug mode to observe its behaviour and debug it. The replay system should be able to simulate the responses from other services from their recorded message logs so that the service in question can be rerun in identical conditions in order to be able to debug it. Obviously selection of what to replay is a critical programmer choice in ensuring that such replays do not hide faults in other services.

Debugging tool repository This is really a subsection of the *version control system* where code is stored for test harnesses, etc., to be used with particular services. One such debugging tool is a separate plug-in to decode messages from each service into a human readable format which the debugger can use in presenting messages to the programmer. As much as possible, these plug-ins should provide at least one output format which is text based and therefore comparable with *diff* or similar tools. The version control system must maintain the relationships between the test harnesses, plug-ins and services. This repository may also need to be able to store executables, in case the service provider will not provide source code.

System Monitor This component should log aspects of the system for later analysis, and use in system optimisation and intervention. Many system faults occur at boundary conditions, e.g. when the disks become too full performance usually falls exponentially. Similarly it is useful to record the number of invocations of each service because some faults such as memory leaks emerge only after a certain number of usages between restarts of the service. Not only can this data be used to provide information to the programmer for debugging, it can also be used to automate some work-arounds such a restarting a service before it becomes unserviceable due to accumulated corruption, or to do a disk clean up. Each service should provide an interface for the system monitor to request its unique vital statistics e.g. for a database: number of queries, number of disk space allocations. There should be a standard for this data, and a core set of this type of data for all services e.g number

of invocations, average time to commence and to complete service. There should also be system interfaces provided by each service to perform a self diagnostic, to acquiesce[9], and to shutdown. This allows the System Monitor to cleanly restart services.

4.3 Implementation

As would be clear by reading the above, the implementation of such a system is significant but not infeasible. For example, many of the components such as the source code / version control system could be built using existing open source software. It will be the degree and clarity of their integration which most influences the utility of the overall system as these functions necessarily interact. However, as much as possible, the integration should be achieved through loose coupling. The implementation will also have to guard against excessive resource consumption, both by being locally efficient, and by imposing minimum costs when not being used. Furthermore, the dynamic and extensible nature of SOAs suggest that any debugger must itself be as extensible as possible, and that any debugger implementation or design should attempt to maximise this extensibility.

5 Existing "SOA" Debuggers

While several companies and systems offer what they call SOA debuggers, the current market shows scant evidence of so-called SOA debuggers providing features specific to SOA rather than just being distributed debuggers. A typical example of this is Microsoft's C# cluster debugger[Cor09, Cor10] which requires the service to be debugged to be linked to specific libraries and appears to provide little more than remote distribution, breakpointing and debugging of the service and client components.

There are many distributed debuggers and many of their features are directly useful in producing a debugger for an SOA environment. In particular, Allinia Software's parallel code debugger[O'C09] allows the programmer to form category groups based on particular trace characteristics. However, few, if any, debuggers are built primarily for debugging SOA environments.

IBM's Websphere offering, does however provide SOA specific debugging funtionality with its "Websphere Business Monitor Debugger"[IBM10] in which choreographed interactions can be debugged by examining the message queues and editing their content.

It is the author's conjecture that just as a system qualifies as an SOA due to the combination of a critical mass of different features that appear in other distributed systems and are strongly integrated to form an SOA, no single debugging feature including ESB control integration will make a debugger an SOA debugger; rather it will be the gestalt of well integrated features that lend themselves to an SOA environment that will make a debugger truly Service Oriented.

[9]To complete all extant transactions but refuse all new requests, returning an acquiesced indication once idle. This is non-trivial because they should not refuse new work from other's extant transactions.

6 SOAs in Defence

6.1 Issues

While the vast majority of issues covered in this paper are common across all business enterprises including Defence there are some issues and risks worth specifically highlighting:

- Defence uses some extremely complex legacy systems, some written in languages no longer in common use. Wrapping these in a messaging interface may be difficult.
- A significant proportion of Defence software is "not written here" or is embedded, or both.
- Some Defence software has extreme performance requirements.
- Some Defence software has mission critical correctness and reliablity requirements that can ultimately mean life or death.
- Some Defence environments provide only impoverished, highly constrained network bandwidth.
- Computing resources may be deployed to remote locations and theatres of war.

6.2 Recommendations

The following practices should assist Defence to maximise its value from SOA:

- Choose an SOA infrastructure that allows Defence to alter and most specifically extend the information conveyed to the ESB or middleware, and which in particular can be altered to add vector timestamps. Other areas where extensibility is important for debugging and logging is in program invocation to be able to start debuggers and perform sentinel functions prior to invocation and after termination. Overall many functions in the specification given earlier rely on being able to alter significant parts of the system software infrastructure. All of this points to selecting as much open source based infrastucture as possible, or at least proprietary software that provides sufficient hooks to allow plugins to be built and used to customise component behaviour and provide the extra functionality required.
- Defence needs to define a spectrum of "SOAness" from full SOA through SOA compatable to SOA incompatable in order not only to have clear dialogs about existing and future software systems, but also to be certain that systems with extreme performance requirements are not crippled by SOA overhead yet are still designed or wrapped to fit into the SOA infrastructure to the maximum extent possible. In designing and implementing the debugging system described above, particular attention should be paid to how to instrument systems with high throughput and response time requirements without overly weighing them down. Establishing standard design patterns for interconnection between the various types of subsystems would help greatly in overall reliability, maintainability and debugging.

- The ability to automatically generate substantial parts of a bug report to forward to software providers is a significant capability for increasing and maintaining service reliability. Provided people do not fall into the trap of thinking that such data is sufficient for a bug report in and of itself, such a facility can greatly increase maintenance productivity.

- Measuring the actual usage of software services will not only assist in determining services' true customers but will also help in determining the true demand for particular services so that resources can be allocated to areas of highest demand.

- Because orchestration is less variable than choreography, stronger reliability assessments can be made about components in the former and debugging orchestrations is easier so Defence should prefer the former solution whenever a choice is possible.

- The possibility of deployment to impoverished network environments means that the debugging and logging system must have a mode in which, rather than returning logs to a central point for correlation as would usually be the case, log fragments are stored locally and only retrieved and correlated when there is a specific need to do so for a particular run or when there is spare bandwidth available. Otherwise, for network constrained environments such logs should be retrieved and correlated post deployment.

- Defence should take particular care to build a representative test infrastructure so that new services and higher level applications can be thoroughly tested and debugged before they progress to deployment. This also requires a working governance regime that assesses reliability, availability, and fitness for purpose and provides a defined process for progressing from design to implementation, testing and deployment.

- Given the need for remote deployment, training for end users should be provided to encourage the lodging of sensible bug reports from the field.

- To maximise field reliability, Defence's SOA architecture policy should include the use of standard Defence designed node templates that allow new "clean" computing and storage nodes to be installed on standard hardware in the field and allow non-deployed programmers to migrate services and applications to those nodes while ensuring that key data is synchronised back into the Defence core SOA infrastructure on a regular basis. This includes logs to facilitate remote debugging from safe locations.

7 Conclusions

Many distributed debuggers exist but few, if any, debuggers designed specifically for full scope SOA debugging currently exist. There are several reasons for this including:

- SOA debuggers cannot be written as stand alone pieces of software since they must integrate strongly with the SOA in which they operate.

- Much of the debugging functionality cannot reside in the debugger itself but must be incorporated into various components of the SOA including the ESB, logger, monitor, etc.
- SOA is still a relatively new concept and little experience with SOAs has been shared.

As such, the contribution of this paper is to point out some prior knowledge which would transfer well to the SOA context such as Mattern[Mat89] and Fidge's[Fid96] independent proposals to use Lamport Clocks to distinguish between different possible parallel executions of distributed code. We give an explicit and more general specification of Category Partitions first used in Allinia software's parallel debugger. We also provide a specification for an SOA debugger that integrates many potential features that assist with debugging distributed and particularly SOA which we believe will provide much greater value when used in combination. For instance the use of vector timestamps to recognise trace instances which are essentially the same and the facility to classify traces according to category partition (a nomenclature which we have introduced). This specification does not mandate particular implementations, but does give some guidance towards how implementation could be achieved.

Notably, it seems that working SOA debuggers require significant integration into the SOA and this indicates that open source SOA architectures may well be the most amenable to hosting truly Service Oriented debuggers. This integration requirement suggests that SOA debugers should be written to be as modularly extensible as possible, perhaps via the use of plug-ins.

While the effort to build an integrated SOA debugging and logging system is significant, so are its benefits. With judicious selection of the order of feature implementation large benefits could be gained early in the development with the final features magnifying earlier return on investment.

Acknowledgements

The author would like to thank Damian O'Dea and Derek Henderson for their helpful review comments.

References

Aga02. David J. Agans. *Debugging: the 9 indispensable rules for finding even the most elusive software and hardware problems.* AMACOM, a division of American Management Association, 1601 Broadway, New York, NY 10019, 2002.

Arm07. *LWD 5-1-1, Staff Officer's Guide 2007*, chapter 8. Commonwealth of Australia, Department of Defence, 2007.

Bec99. Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison Wesley Professional, 1999. ISBN-10: 0-201-61641-6.

CIO10. CIOG. Single Information Environment (SIE): Architectural Intent 2010. Technical Report DPS: DEC013-09, Commonwealth of Australia, Department of Defence, May 2010.

CoA11. Department of Defence Commonwealth of Australia. Chief information officer group instruction no. 1/2011. Departmental dissemination., May 2011.

Cor09. Microsoft Corporation. Using the C# Cluster-SOA debugger for Windows HPC 2008 R2 (Visual Studio 2010), 2009. http://msdn.microsoft.com/en-us/library/gg604920.aspx Viewed 2011-06-06.

Cor10. Microsoft Corporation. Using the HPC cluster debuggers for SOA and MPI applications, August 2010. http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=23213 Viewed 2011-08-17.

DH04. Schahram Dustdar and Stephan Haslinger. Testing of service-oriented architectures - a practical approach. In *Net.ObjectDays*, pages 97–109, 2004.

Fid96. C. J. Fidge. Fundamentals of distributed system observation. *IEEE Software*, 13(6):77–83, November 1996.

Fid98. C. J. Fidge. A limitation of vector timestamps for reconstructing distributed computations. *Information Processing Letters*, 68(2):87–91, 1998.

IBM10. IBM. IBM Education Assistant IBM WebSphere Business Monitor Version: V7.0 Debugger, 2010. http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/index.jsp?topic=/com.ibm.iea.wbmonitor_v7/wbmonitor/7.0/Debugger.html Viewed 2011-08-17.

Jos07. Nicolai M. Josuttis. *SOA in Practice*. /Theory/In/Practice. O'Reilly, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2007. ISBN-13: 978-0-596-52955-0.

Mat89. Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

MDL87. H.D. Mills, M Dyer, and R.C. Linger. Cleanroom software engineering. *IEEE Software*, 4(5):19–25, September 1987. ISSN: 0740-7459.

NL05. Eric Newcomer and Greg Lomow. *Understanding SOA with Web Services*. Addison-Wesley, Boston, MA, 2005.

Oas06. Reference model for service oriented architecture, October 2006. http://docs.oasis-open.org/soa-rm/v1.0/.

O'C09. Mark O'Connor. Parallel debugging is easy. White Paper, Allinea Software Ltd, The Innovation Centre, Warwick Technology Park, Gallows Hill, Warwick, CV34 6UW, UK, 2009. http://www.allinea.com/index.php?page=84 Viewed 2011-06-06.

PMM93. J.H. Poore, Harlan D. Mills, and David Mutchler. Planning and certifying software system reliability. *IEEE Software*, 10(1):88–99, January 1993.

Qos. Wikipedia entry for Quality of Service. http://en.wikipedia.org/wiki/Quality_of_service Viewed 27 June 2011.

SOA06. Wikipedia entry for Service-Oriented Architecture, 2006. http://en.wikipedia.org/wiki/Service-oriented_architecture Viewed 2011-05-17.

THIS PAGE IS INTENTIONALLY BLANK

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION DOCUMENT CONTROL DATA

1. CAVEAT/PRIVACY MARKING

2. TITLE	3. SECURITY CLASSIFICATION
Debugging and Logging Services for Defence Service Oriented Architectures	Document (U) Title (U) Abstract (U)
4. AUTHOR	5. CORPORATE AUTHOR
Michael Pilling	Defence Science and Technology Organisation PO Box 1500 Edinburgh, South Australia 5111, Australia

6a. DSTO NUMBER	6b. AR NUMBER	6c. TYPE OF REPORT	7. DOCUMENT DATE
DSTO–TR–2664	AR-015-224	Technical Report	February 2012

8. FILE NUMBER	9. TASK NUMBER	10. TASK SPONSOR	11. No. OF PAGES	12. No. OF REFS
2011/1195600/1	CDG 07/355	DG Integrated Capability Development	17	20

13. URL OF ELECTRONIC VERSION	14. RELEASE AUTHORITY
`http://www.dsto.defence.gov.au/publications/scientific.php`	Chief, Command, Control, Communications and Intelligence Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for Public Release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SOUTH AUSTRALIA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS

No Limitations

18. DSTO RESEARCH LIBRARY THESAURUS

Debugging
Distributed Computing
Software Maintenance
Service Oriented Architecture
Software Tools
Software Architecture

19. ABSTRACT

While often thought of as a "Dark Art", debugging is nevertheless a necessary part of fielding quality computing systems which can and should be done systematically. Service Oriented Architectures (SOAs) show great promise but also represent one of the most challenging environments in which to debug system services. In addition to all the issues of distributed and paralled debugging, SOAs introduce the complexity of significant parts of one's programs being provided by others. This paper examines the features of SOAs that complicate debugging and shows how integrated logging is an essential part of finding the cause of service failures. We draw on Agan's work in developing systematic strategies for debugging to generate system features that are necessary or helpful for debugging in an SOA environment. These are in turn used to specify requirements for a debugging system integrated into the fabric of the SOA. We argue that deep integration is necessary to produce significant debugging efficiency improvement in an SOA environment and provide some recommendations for Defence in this area.